ISSN 0280–5316
ISRN LUTFD2/TFRT--5578--SE

A Real Time Control Language for the Pålsjö Environment

Anders Blomdell

Department of Automatic Control
Lund Institute of Technology
May 1997

Department of Automatic Control
Lund Institute of Technology
Box 118
S-221 00 Lund Sweden

Document name
MASTER THESIS

Date of issue
May 1997

Document Number
ISRN LUTFD2/TFRT--5578--SE

Author(s)
Anders Blomdell

Supervisor
Karl Johan Åström

Sponsoring organisation

Title and subtitle
A Real Time Control Language for the Pålsjö Environment

Abstract

Traditionally computer based real-time control systems have often been coded from scratch, involving a substantial effort. Recently the adoption of domain specific frameworks has made it easier to implement real-time control systems. Pålsjö is one such framework implemented at the Department of Automatic Control in Lund.

During the construction of the Pålsjö real-time control system framework, it was obvious that ordinary programming languages are not an optimal choice for the implementation of control algorithms. It was also apparent that the algorithms got intermixed with framework specific code. A new language was therefore proposed and implemented. A properly designed language with data-types and other abstractions aimed at control engineers simplifies the coding of algorithms. Since the code executed by the framework is generated by a compiler, the structures in the framework can be designed for maximum performance instead of ease of use. The reuse of algorithms in a new or modified framework is also simplified since only a new back-end for the compiler is required.

The main focus of the thesis are the syntactic and semantic elements of the proposed language, but a brief introduction to the compiler structure is also given as an aid for the implementation of new back-ends.

Key words
Real-Time Control, Control Algorithm, Compiler, Programming Language

Classification system and/or index terms (if any)

Supplementary bibliographical information

ISSN and key title
0280–5316

ISBN

Language
English

Number of pages
29

Recipient's notes

Security classification

Department of Automatic Control
Lund Institute of Technology
Box 118
S-221 00 LUND
Sweden

ISSN 0280–5316
ISRN LUTFD2/TFRT--5578--SE

Lund, May 1997

Contents

1. Introduction

Construction of digital control systems is done by iterations of the two distinct phases design and implementation. The design phase, that consists of the three tasks modeling, control design and simulation, has adequate computer support. The second phase that includes coding, compilation, experimentation, debugging and modification, still lack tool support for the control engineering concepts used in the first phase. Several attempts has been done to remedy this. One possibility is to base all work on a graphical system representation that is used for simulation [MathWorks, 1996] and automatic generation of real time code. This works quite well for simple control tasks consisting of a few loops but not so well for more complicated tasks that also includes logic and sequencing. To implement a good real time system it is also essential to have a more detailed representation of the interaction of the real time tasks than can be provided by a block diagram for a control system.

A good experimental environment should also be interactive. It should thus be possible to change parameters and even structure on-line. This thesis describes the **pal** language used in an attempt to develop a system that fulfills the goals above. The idea has been to investigate if a flexible interactive environment can be implemented with a reasonable effort.

1.1 Pålsjö

Pålsjö [Eker and Blomdell, 1996] is a software environment for development of real-time control systems. It is a framework for construction and execution of control applications which focus on rapid prototyping of control systems. It exploits the traditional metaphor of control blocks connected to each other. The designer off-line defines a set of blocks which later at run-time can be instantiated and connected to form a control system. Since timing is critical in many control systems, special effort has been made to help minimize computational delays of algorithms.

The Pålsjö system consists of two main parts; the compiler described in this thesis and a run-time system. The compiler translates blocks written in PAL into C++-code. The C++-code is then compiled and linked with the run-time system. The run-time systems provides a text interface for the user and a network interface for data exchange. This allows a configuration where the controller executes on one machine and data presentation is executed on another machine. During runtime all kinds of modifications to controller structure and parameters can be done without stopping the system.

1.2 pal

During the initial phase of the Pålsjö design, algorithms were written in C++ [Lippman, 1989], but it was soon apparent that high-level programming languages lacked mechanisms to efficiently isolate algorithms from even minor architectural changes

of the underlying run-time system, necessitating many rewrites of all algorithms as the platform evolved.

To protect the investment in algorithms, the language **pal** (Pålsjö Algorithm Language) was introduced. The main focus during the design of **pal**, has been that the notation should be easy to read and make the semantic gap between control theory and the implementation of algorithms as small as possible. It's not intended to be a full fledged language for everyday programming, which means that things like pointers and bit manipulations are intentionally omitted from the language, while polynomials, matrices and Grafcets are supported.

An added benefit of a specialized language, is that it's relatively easy to create a new back-end, so the same algorithm can be reused in a new framework.

1.3 Experiences so far

Pålsjö has now been used experimentally in several projects:

- Control of inverted pendulums.
- Integrated Control and Diagnostics Using Robust Control Methods [Åkesson, 1996].
- General adaptive regulators.
- Hybrid Control of a Double Tank System [Malmborg and Eker, 1997].
- Fuzzy control (laboratory exercise in Control System Synthesis).
- Auto-tuning of Robot Servo (course project in Adaptive Control/Real-Time programming).

During the experiments it has been demonstrated that a specialized language tailored for control engineers helps to focus the effort on control issues instead of low level programming details. The system is useful after a very brief introduction, and clearly shows that the introduction of yet another language helps to further improve the control engineers productivity.

The Pålsjö system runs on Sun/Solaris, VME-m68k, and Windows NT. The compiler is currently only available on Solaris.

1.4 Future enhancements

Since the main focus of **pal** is to transform algorithms to other representations, it's important that as much of the design as possible be propagated. Since comments are often used to augment algorithms, they should preferably be preserved in the transformation. Unfortunately this is currently not the case, since comments are stripped off in the compiler's initial lexical analyzer. The right way to go, is to make comments an integral part of the **pal** language.

The analysis of Grafcets should be rewritten, since it currently doesn't handle some legal Grafcets (the code generation is believed to be correct though).

In an attempt to formally capture design constraints of algorithms, work is currently in progress to add the notion of contracts [Helm *et al.*, 1990] to the language.

1.5 Outline of this report

The report will start with a few simple control engineering examples, then a description of **pal** program structure is followed by a chapter about Grafcet. The appendices covers compiler usage, syntax and the nitty-gritty details of the actual compiler implementation.

1.6 Inspiration and acknowledgments

The **pal** language constructs has many different roots, some things that immediately comes to mind are: the module concept mimics the one in Modula-2 [Wirth, 1985], the way to return function results is borrowed from Eiffel [Meyer, 1988], the way to express Grafcet [David, 1995] charts comes from the IEC 1131 [Int, 1992] standard and the overall look has its roots in Algol [Rutishauser, 1967]. The compiler implementation has been greatly simplified by the use of the well integrated **cocktail** suite of compiler-compiler tools [Grosch, 1991].

Special thanks go to Johan Eker and Anders Robertson that gave valuable critic and suggestions during the initial design of **pal**.

2. Examples

This section will present a few building blocks that shows how common control engineering algorithms can be expressed in **pal**.

2.1 PID controller

A simple PID controller with anti-windup is implemented in the following **pal** module.

```
module ExamplePID;
  block PID
    y, yref, v : input real;
    u : output real;
    K, Ti, Td, Tr, N : parameter real;
    P, I, D, e, yold : real;
    h : sampling interval;
    d1 = Td / (Td + N * h);
    d2 = K * N * d1;
    i1 = h * K / Ti;
    w1 = h / Tr;
    forward begin
      e := yref - y;
      P := K * e;
      D := d1 * D + d2 * (yold - y);
      u := P + I + D;
    end forward;
    backward begin
      yold := y;
      I := I + i1 * e + w1 * (v - u);
    end backward;
  end PID;
end ExamplePID.
```

2.2 Summing block

A summing block that can handle any number of inputs is shown in the following **pal** code.

```
module ExampleSum;
  block Sum
    n : dimension;
```

```
    in : array [1..n] of input real;
    out : output real;
  forward
    tmp : real;
    i : integer;
  begin
    tmp := 0.0;
    for i := 1 to n do
      tmp := tmp + in[i];
    end for;
    out := tmp;
  end forward;
end Sum;
end ExampleSum.
```

2.3 State-space controller

A single input single output (SISO) state-space controller can be implemented as follows:

```
module ExampleStateSpace;
  block StateSpace
    n : dimension;
    A : parameter matrix [1..n, 1..n] of real;
    B : parameter matrix [1..n, 1..1] of real;
    C : parameter matrix [1..1, 1..n] of real;
    D : parameter matrix [1..1, 1..1] of real;
    x : matrix [1..n, 1..1] of real;
    u : input real;
    y : output matrix [1..1, 1..1] of real;
  forward begin
    y := C * x + D * u;
  end forward;
  backward begin
    x := A * x + B * u;
  end backward;
end StateSpace;
end ExampleStateSpace.
```

3. Program structure

3.1 Data

Data is the common name of all the items that an algorithm operates on. Every piece of data has an associated type that determines what can be done with that specific item. In addition to the type, an item may have associated modifiers that affect what can be done with it.

Scalar types

Boolean A boolean value is one of the logical truth values true or false. A boolean value is returned by:

- The predefined identifiers **true** or **false**.
- The logical operators **and**, **or** or **not** applied to boolean operands.
- The relational operators < (less than), <= (less than or equal to), <> (not equal to), = (equal to), >= (greater than or equal to) or > (greater than) applied to **integer**, **real** or **polynomial** operands.
- A call to a boolean function.
- Reference to a boolean variable.

Integer An integer value is a natural number that falls within some implementation imposed limits. An integer value is returned by:

- The operators *, **div**, **mod**, + or − applied to integer operands.
- A call to an integer function.
- Reference to an integer variable.

Real A real value is a real number that falls within some implementation imposed limits. An real value is returned by:

- The operators *, /, + or − applied to real operands.
- A call to a real function.
- Reference to a real variable.

Dimension A dimension variable is an integer which gets its value when an instance of a block is created. Inside algorithms it is used as a constant integer.

```
n : dimension;
...
for i := 1 to n do
```

Sampling interval The sampling interval (in seconds) of a specific block can be accessed by declaring a sampling interval variable and use it as a real.

```
h : sampling interval;
...
i := i + h * K * e / Ti;
```

Aggregate types

Array An array is a bounded sequence of elements of some type. The bounds can be any integer expression that can be evaluated where the array is declared. The only operations supported on arrays are assignment and subscripting.

```
in : array [1..n] of input real;
...
tmp := in[1];
for i := 2 to n do
    tmp := tmp + in[i];
end for;
```

Matrix A matrix is a two-dimensional array of reals. A matrix value is returned by:

- The operators $*$, $+$ or $-$ applied to matrix operands.
- The operator $*$ applied to a scalar and a matrix operand.
- Reference to a matrix variable.

```
x : matrix [1..n, 1..1] of real;
A : parameter matrix [1..n, 1..n] of real;
B : parameter matrix [1..n, 1..1] of real;
...
x := A * x + B * u;
```

Polynomial A polynomial is a one-dimensional array of reals. A polynomial value is returned by:

- The operators $*$, **div**, **mod**, $+$ or $-$ applied to polynomial operands.
- The operator $*$ applied to a scalar and a polynomial operand.
- Reference to a polynomial variable.

A polynomial can be evaluated in a specific point by treating it as a function of a single real argument.

```
T, Am, Ao, B : polynomial [n] of real;
...
T := Am(1.0) / B(1.0) * Ao;
```

Modifiers

Input The **input** modifier is used in block or procedure declarations to indicate that a variable is an input. An input variable may not be assigned a value.

```
in : input real;
```

Output The **output** modifier is used in block or procedure declarations to indicate that a variable is an output.

```
out : output real;
```

Parameter The **parameter** modifier is used in block declarations to indicate that a variable is a parameter. A parameter is a variable that only changes when the system is reconfigured.

```
par : parameter real;
```

Derived parameter

Derived parameters are named expressions that only depends on parameters, constants and the sampling interval. They are reevaluated every time a parameter or the sampling interval of the block changes. Since they are calculated only when necessary, use of them can lower the total workload on the system. Their type is deduced from the expression and should not be declared.

```
K, Ti : parameter real;
h : sampling interval;
...
bi = h * K / Ti;
```

3.2 Expressions

Expressions describe how the calculation of new data values should be performed. What calculations can be done with a specific item, depend on its type and modifiers.

Unary expressions

Unary expressions only involve one operand (either an expression or an item of data). In **pal** three kinds of unary expressions are possible:

- Referencing a data item.
- Arithmetic negation (e.g. −pi).
- Boolean negation (e.g. **not** bad).

Binary expressions

Binary expressions involve two operands. There are three distinct kinds of binary expressions:

- Arithmetic expressions (i.e. +, −, ∗, /, **mod** or **div**), that returns an arithmetic result.
- Relational expressions (i.e. <, <=, <>, =, >= or >), that returns a boolean result.
- Polynomial evaluation, that returns the value of the polynomial when evaluated in a specific point.

Other expressions

Other expressions involves a variable number of operands.

- Subscripting (e.g. x[i]), that returns a specific part of an array, matrix or polynomial.
- Functions calls, that return a result of the kind specified in the function declaration.

3.3 Statements

Every useful algorithm generate output values that eventually influence the environment. The values are generated by the execution of one or more actions described by **statements**. Statements are either **simple** (e.g. assignment) or **compound**.

Simple statements

Assignment The most fundamental statement is the **assignment statement**. It specifies that a variable should assigned the value of an expression.

```
in : input real;
out : output real;
K : parameter real;
...
out := K * in;
```

Procedure call To make algorithms easier to understand and maintain, **pal** has the concept of **procedures**. A procedure is called by a statement containing the name of the procedure followed by a parenthesized list of its actual arguments. How parameters are passed, depend on what modifiers (**input** or **output**) are used in the procedure declaration.

```
P();
```

Compound statements

If statement The if statement is used to select between a number of disjoint statement sequences (branches). At most one of its branches can be selected each time the if statement is executed. The statements in the first branch that satisfies

its condition will be executed. If no condition is satisfied, the else branch will be chosen.

```
if value < min then
  result := min;
elsif value > max then
  result := max;
else
  result := value;
end if;
```

For statement The for statement is used to execute a sequence of statements a specified number of times. Often the bounds of the loop variable are determined by block **dimension** variables.

```
n : dimension;
in : array [1..n] of input real;
out : output real;
...
for i := 1 to n do
  out := out + in[i];
end for;
```

3.4 Procedures and functions

Procedures

Procedures are used to group together statements in the algorithm. Their objective is twofold:

- The statements can be given a descriptive name.
- Statements can be reused in more than one place in the algorithm.

```
procedure MinMax(
  min : output array [1..n : integer] of real;
  max : output array [1..n] of real;
  a1 : input array [1..n] of real;
  a2 : input array [1..n] of real
);
  i : integer;
begin
  for i := 1 to n do
    if a1[i] < a2[i] then
      min[i] := a1[i];
      max[i] := a2[i];
    else
```

```
        min[i] := a2[i];
        max[i] := a1[i];
      end if;
    end for;
  end MinMax;
```

Functions

Functions are like procedures, but in addition they return a value. In every function there is a reserved variable **result** that is of the same type as the function return type. The **result** variable can be used as any other variable of its type.

```
function Min(
  a1 : input array [1..n : integer] of real
) : real;
  i : integer;
begin
  result := a1[1];
  for i := 2 to n do
    if result < a1[i] then
      result := a1[i];
    end if;
  end for;
end Min;
```

3.5 Forward and backward

To accommodate control engineering requirements [Åström and Wittenmark, 1990], there are two predeclared block procedures, the **forward** procedure to calculate outputs and the **backward** procedure to update internal states after all other blocks have run their **forward** procedures. A typical example is a simple **PI**-controller with anti-windup.

```
block PI
  y, yref, v : input real;
  u : output real;
  K, Ti, Tr : parameter real;
  P := 0.0, I := 0.0, e : real;
  h : sampling interval;
  forward begin
    e := yref - y;
    P := K * e;
    u := P + I;
  end forward;
  backward begin
    I := I + h * K / Ti * e + h / Tr * (v - u);
  end backward;
```

```
end PI;
```

3.6 Modules and blocks

Modules

Modules is the top-level structuring concept in **pal**. Inside a module **blocks**, **procedures** and **functions** are declared. Modules are used to package blocks that are somehow related to each other. Things declared in one module are not visible in other modules unless they are explicitly imported.

Blocks

Blocks correspond to the black boxes that control engineers uses as their main abstraction view. A block's main characteristic is that it has **inputs** and **outputs**. It may also have **parameters** that reflects the parameters used in algorithms (e.g. the gain K), **dimensions** and **derived parameters**.

```
module PController;
  block P
    y, yref : input real;
    u : output real;
    K : parameter real;
    forward begin
      u := (yref − y) ∗ K;
    end forward;
  end P;
end PController.
```

4. Grafcet

Grafcet is a convenient way to express sequences and in **pal** they can be used to ensure the proper sequencing of algorithms as well as sequence control of external systems.

4.1 Steps

Steps represent a particular state of a sequence, and are roughly equivalent to a state of a state-machine, except that more than one step may be active at the same time. A sequence always starts in its mandatory **initial step**.

```
initial step Init;
  pulse activate V2off;
end Init;
step HeatOn;
  activate Qon;
end HeatOn;

step HeatOff;
  pulse activate Qoff;
end HeatOff;
```

As long as a step is active, its **action** associations are evaluated to determine if the action should be run. The associations to actions can be of nine different kinds, namely:

- `activate` «identifier» – the action is run while the step is active.
- `pulse activate` «identifier» – the action is run once.
- `limit` «expression» `activate` «identifier» – the action is run while the step is active, until the specified time has elapsed.
- `delay` «expression» `activate` «identifier» – the action is delayed until the specified time has elapsed, and is then run while the step is active.
- `store activate` «identifier» – the action is run until explicitly `reset`.
- `store limit` «expression» `activate` «identifier» – the action is run until the specified time has elapsed, and then has to be explicitly `reset` until any other timed association can be legally activated.
- `store delay` «expression» `activate` «identifier» – the action is delayed until the specified time has elapsed, and is then run until explicitly `reset`.
- `delay` «expression» `store activate` «identifier» – if the step is still active after the specified delay has elapsed, the action is run until explicitly `reset`.
- `reset` «identifier» – a previously `stored`, `store delayed`, `delay stored` or `store limited` action is reset.

4.2 Actions

Actions contain the code that should be executed when it has been activated by some step. As long as it is active, the action executes once every sampling period.

```
action Qon;
begin
  Heater := Low;
end Qon;

action Qoff;
begin
  Heater := false;
end Qoff;
```

4.3 Transitions

Transitions control the activation and deactivation of steps. A transition is fired if all the steps leading into it are active and the associated condition is fulfilled. When a transition is fired, the steps preceding it are deactivated, and the steps following it are activated.

```
transition from HeatOn to HeatOff when T >= Tref;

transition from HeatOff to HeatOn when T < Tref;
```

4.4 Example: A boiler process

The following **pal** code implements a grafcet for the control of a simple boiler.

```
module Boiler;
  block Boiler
    Start, Low, High : input boolean;
    Heater, V1, V2 : output boolean;
    T : input real;
    Tref : parameter real;

    initial step Init;
      pulse activate V2off;
    end Init;

    step HeatOn;
      activate Qon;
    end HeatOn;

    step HeatOff;
      pulse activate Qoff;
    end HeatOff;

    step Fill;
```

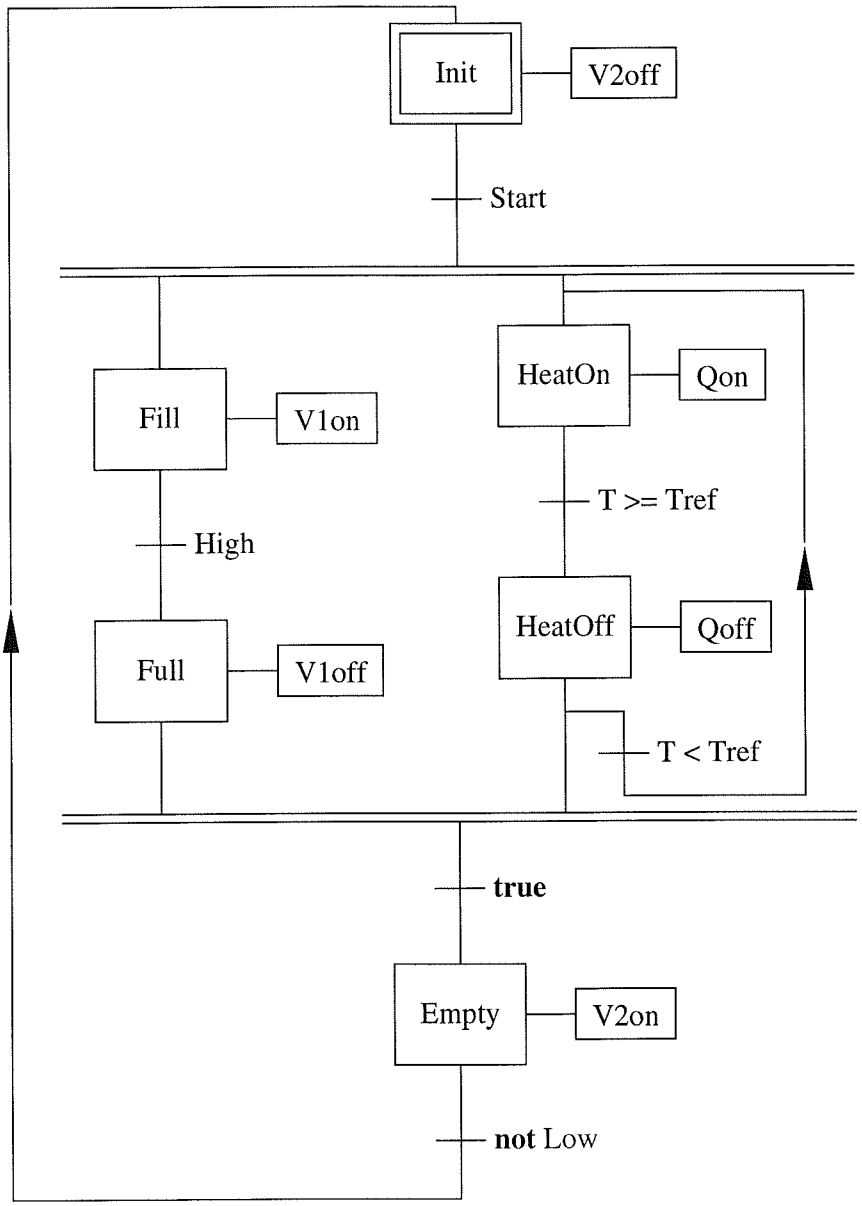

Figure 4.1 Grafcet describing control of a simple boiler

```
  pulse activate V1on;
end Fill;

step Full;
  pulse activate V1off;
end Full;

step Empty;
  activate V2on;
end Empty;

action V1on;
begin
  V1 := true;
end V1on;
```

```
    action V1off;
    begin
      V1 := false;
    end V1off;
    action V2on;
    begin
      V2 := true;
    end V2on;
    action V2off;
    begin
      V2 := false;
    end V2off;
    action Qon;
    begin
      Heater := Low;
    end Qon;
    action Qoff;
    begin
      Heater := false;
    end Qoff;
    transition from Init to Fill, HeatOn when Start;
    transition from HeatOn to HeatOff when T >= Tref;
    transition from HeatOff to HeatOn when T < Tref;
    transition from Fill to Full when High;
    transition from HeatOff, Full to Empty when true;
    transition from Empty to Init when not Low;
  end Boiler;
end Boiler.
```

5. Bibliography

ÅKESSON, M. (1996): "Integrated control and diagnostics using robust control methods." In *EURACO Workshop Robust and Adaptive Control of Integrated Systems.* Herrsching, Germany.

ÅSTRÖM, K. J. and B. WITTENMARK (1990): *Computer Controlled Systems—Theory and Design*, second edition. Prentice-Hall, Englewood Cliffs, New Jersey.

DAVID, R. (1995): "Grafcet: A powerful tool for specification of logic controllers." *IEEE Transactions on Control Systems Technology*, **3:3**, pp. 253–268.

EKER, J. and A. BLOMDELL (1996): "A structured interactive approach to embedded control." In *The 4th International Symposium on Intelligent Robotic Systems.* Lisbon, Portugal.

GROSCH, J. (1991): "Toolbox introduction." Technical Report 25. Gesellschaft für Mathematik und Datenverarbeitung mbH, Vincenz-Prießnitz-Str. 1, D-7500 Karlsruhe.

HELM, R., I. M. HOLLAND, and D. GANGOPADHYAY (1990): "Contracts: Specifying behavioral compositions in object-oriented systems." In *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages and Applications*, pp. 169–180.

INTERNATIONAL ELECTROTECHNICAL COMMISSION (1992): *IEC 1131-3, Programmable controllers, Part 3: Programming languages.*

LIPPMAN, S. B. (1989): *C++ Primer.* Addison-Wesley.

MALMBORG, J. and J. EKER (1997): "Hybrid control of a double tank system." In *CCA'97.* Hartford, CT. (in submission).

MATHWORKS (1996): *Using Simulink.* The MathWorks Inc, South Natick, Massachusetts.

MEYER, B. (1988): *Object-Oriented Software Construction.* Prentice Hall.

RUTISHAUSER, H. (1967): *Description of Algol 60.* Handbook of Automatic Computation, Vol. 1a. Springer, Berlin.

WIRTH, N. (1985): *Programming in Modula-2*, 3d edition. Springer, New York.

A. Using the compiler

After a control algorithm has successfully been translated to a **pal** module, it is time to compile it. The compilation can be considered to be done in two distinct steps; **analysis** and **code generation**. The compiler is invoked by the command `pal <options> <files>`. The supported options are:

- `-pretty` generate a pretty printed text-version of the module.
- `-palsjo` generate C-files that can be compiled and linked into the pålsjö run-time system.
- `-html` generate a pretty printed version of the module with HTML markup commands embedded.
- `-tex` generate a pretty printed version of the module with TeX markup commands embedded.
- `-fig` generate a fig version of all grafcets in a module.

To invoke the analysis step only a filename is necessary, but to generate any useful output at least one of the code-generation options above has to be given.

B. Reserved words and syntax

B.1 Reserved words

The reserved symbols and words in PAL are:

(	=	dimension	initial	pulse
)	>	div	input	real
*	>=	do	integer	reset
+	[	downto	interval	sampling
,	]	else	limit	step
−	action	elsif	matrix	store
.	activate	end	mod	then
..	and	external	module	to
/	array	false	not	transition
:	backward	for	of	true
:=	begin	forward	or	when
;	block	from	output	while
<	boolean	function	parameter	
<=	declare	if	polynomial	
<>	delay	import	procedure	

B.2 Syntax

Typographical notation

In the following Extended Backus-Naur Form (EBNF) syntax diagrams the following typographical elements are used:

- Terminal symbols (e.g. `end`).
- Non-terminal symbols (e.g. «module»).
- Grouping (e.g. ⟦ , «identifier» ⟧).
- Optional parts (e.g. «actuals»?).
- Selection (e.g. ⟦ «expression» ‖ «identifier» : `integer` ⟧).
- Repetition zero or more times (e.g. ⟦ , «identifier» ⟧*).
- Repetition one or more times (e.g. «digit»+).

EBNF syntax

letter ⇒
a ‖ b ‖ c ‖ d ‖ e ‖ f ‖ g ‖ h ‖ i ‖ j ‖ k ‖ l ‖ m ‖ n ‖ o ‖ p ‖
q ‖ r ‖ s ‖ t ‖ u ‖ v ‖ w ‖ x ‖ y ‖ z ‖
A ‖ B ‖ C ‖ D ‖ E ‖ F ‖ G ‖ H ‖ I ‖ J ‖ K ‖ L ‖ M ‖ N ‖ O ‖ P ‖
Q ‖ R ‖ S ‖ T ‖ U ‖ V ‖ W ‖ X ‖ Y ‖ Z

digit ⇒ 0 ‖ 1 ‖ 2 ‖ 3 ‖ 4 ‖ 5 ‖ 6 ‖ 7 ‖ 8 ‖ 9

identifier ⇒ «letter» [[«letter» ‖ «digit»]]*

module ⇒ module «identifier» «import»* «declaration»* end «identifier» .

import ⇒ [[import «identifier» [[, «identifier»]]*]]? ;

declaration ⇒
[[block «identifier» «declaration»* end «identifier»
‖ procedure «identifier» («formals») ; «body» end «identifier»
‖ function «identifier» («formals») : «type» ; «body» end «identifier»
‖ forward ; «body» end forward
‖ backward ; «body» end backward
‖ «variables» : «type»
‖ «identifier» = «expression»
‖ [[initial]]? step «identifier» ; «action»* end «identifier»
‖ transition from «steps» to «steps» when «expression»
‖ action «identifier» ; «body» end «identifier»
]]? ;

body ⇒ «declaration»* begin «statement»*

formals ⇒ [[«identifier» : «type» [[; «identifier» : «type»]]*]]?

action ⇒
[[activate «identifier»
‖ reset «identifier»
‖ store activate «identifier»
‖ limit «expression» activate «identifier»
‖ delay «expression» activate «identifier»
‖ pulse activate «identifier»
‖ store delay «expression» activate «identifier»
‖ delay «expression» store activate «identifier»
‖ store limit «expression» activate «identifier»
]]? ;

steps ⇒ «identifier» [[, «identifier»]]*

variables ⇒
«identifier» [[:= «expression»]]? [[, «identifier» [[:= «expression»]]?]]*

type ⇒
boolean
‖ dimension
‖ integer

‖ `real`
‖ `sampling interval`
‖ `array` [«range» ⟦ , «range» ⟧*] `of` [type]
‖ `matrix` [«range» , «range»] `of real`
‖ `polynomial` [«limit»] `of real`
‖ `input` «type»
‖ `output` «type»
‖ `parameter` «type»

range ⇒ «limit» . . «limit»

limit ⇒ ⟦ «expression» ‖ «identifier» : `integer` ⟧

statement ⇒
⟦ ⟦ `declare` «declaration»* ⟧? `begin` «statement»* `end`
‖ «identifier» := «expression»
‖ «identifier» [«actuals»] := «expression»
‖ «identifier» («actuals»?)
‖ `if` «expression» `then` «statement»*
⟦ `elsif` «expression» `then` «statement»* ⟧*
⟦ `else` «statement»* ⟧?
`end if`
‖ `while` «expression» `do` «statement»* `end while` .
‖ `for` «identifier» := «expression» `to` «expression»
`do` «statement»* `end for`
‖ `for` «identifier» := «expression» `downto` «expression»
`do` «statement»* `end for`
⟧? ;

expression ⇒
(«expression»)
‖ «identifier»
‖ «identifier» («actuals»?)
‖ «identifier» [«actuals»]
‖ «constant»
‖ `not` «expression»
‖ - «expression»
‖ «expression» «operator» «expression»

operator ⇒ `and` ‖ `or` ‖ = ‖ <> ‖ < ‖ <= ‖ >= ‖ > ‖ + ‖ - ‖ * ‖ / ‖ `mod` ‖ `div`

actuals ⇒ «expression» ⟦ , «expression» ⟧*

constant ⇒
`true`
‖ `false`
‖ «digit»+
‖ «digit»+ . «digit»* «exponent»?
‖ «duration»

exponent ⇒ ⟦ `e` ‖ `E` ⟧ ⟦ + ‖ - ⟧? «digit»+

duration ⇒ Under construction

C. Compiler implementation

This section is suggested reading for anyone who wants to write or modify a code generator to the **pal** compiler.

The **pal** compiler is implemented using the **cocktail** compiler-compiler tools. Cocktail consists of a number of tools that reads specifications in a number of specialized languages and generates C or Modula-2 code according to those specifications. The cocktail toolkit was originally developed at the Karlsruhe subsidiary of the German National Research Center. The reason for using these tools instead of **yacc** and **lex**, is that the cocktail suite is more powerful, better integrated and generates faster code.

C.1 Scanner

The scanner is the part of the compiler that reads pal programs and translates them into a sequence of tokens that are passed on to later stages of the compiler. This is useful since many syntactic elements in pal consists of more than one character. The scanner specification is found in two different files:

- `pal.rex` that specifies the syntax of identifiers, strings, comments and numerical constants.
- `parser.cg` that specifies the concrete syntax of everything not specified in `pal.rex`.

The specifications in `parser.cg` are then processed by the **cg** program generating output which is merged with the hand-written syntax rules from `pal.rex`. The final result is processed by the **rex** program, thereby generating a lexical analyzer, which is later compiled and linked into the compiler.

C.2 Parser

The parser reads the tokens produced by the scanner and uses them to construct a representation of the program that is later used for semantic analysis and code generation. The parser specification is located in the file `parser.cg`.

In the parser, care has been taken to make all recursive rules left recursive, since in this way the parser can build the syntax tree without stacking up a lot of symbols. After the (left-recursive) rule is finished, the resulting syntax tree is reversed, which can be done in linear time with a constant amount of stack. An example is the if statement, whose concrete syntax is declared as:

```
If = 'if' Expr 'then' then:Stats Elsifs Else 'end' 'if' .
Elsifs = <
```

```
    Elsifs0 = .
    Elsifs2 = Elsifs 'elsif' Expr 'then' Stats .
> .
```

The semantic actions for the if and the elsif part of the if statement are:

```
If = {
  Tree := mIf(
    Expr:Tree,
    ReverseTree(then:Tree),
    ReverseTree(Elsifs:Tree),
    Else:Tree);
} .
Elsifs0 = {
  Tree := mElsifs0();
} .
Elsifs2 = {
  Tree := mElsifs1(
    Elsifs:Tree,
    mElsif(Expr:Tree, ReverseTree(Stats:Tree))
  );
} .
```

Now let's consider what happens when the compiler finds the following **pal** statements:

if e1 **then**
elsif e2 **then**
elsif e3 **then**
elsif e4 **then**
end if;

After scanning the **if** and its associated statements (none in this particular example) by the `If` rule, the `Elsifs` rules are tried, and the first **elsif** is encountered. The only rule that matches is the `Elsifs0`, so we get this situation:

Remaining tokens	**Syntax tree**
elsif e2 then elsif e3 then elsif e4 then end if	Elsifs0

In the next three step the `Elsifs2` rule matches, and we get:

Remaining tokens	**Syntax tree**
elsif e3 then elsif e4 then end if	Elsifs2(e2);Elsifs0
elsif e4 end if	Elsifs2(e3);Elsifs2(e2);Elsifs0
end if	Elsifs2(e4);Elsifs2(e3);Elsifs2(e2);Elsifs0

After this we return to the `If` rule, and the tree is reversed by the call to `ReverseTree`, giving the desired result:

Remaining tokens	Syntax tree
end if	Elsifs2(e2);Elsifs2(e3);Elsifs2(e4);Elsifs0

C.3 Semantic analysis

If the **pal** module is successfully parsed (i.e. no syntax errors are found), the semantic analysis starts by propagating type information to all variables and creating lists of all entities visible in different parts of the module. The specifications how this should be done are given by rules in the file `pal.cg`, these specifications are then processed by the **cg** program to generate code to traverse the syntax tree and generate data structures for this information. The rules in `pal.cg` are dependency checked, so the ordering of rules is not important. For a procedure the visibility calculations are handled by the following rules:

(1) Formals:ObjectsIn := mObjects1(mObjects0(), Object);
(2) Body:ObjectsIn := Formals:ObjectsOut;
(3) ObjectsOut := mObjects1(ObjectsIn, Object);
(4) Formals:Env := mEnv1(Env, Formals:ObjectsOut);
(5) Body:Env := mLocalEnv(Formals:Env, Body:ObjectsOut);

Let's analyze what these rules does when presented the following example module:

```
module M;
  block Sample
    r : real;
    procedure P(
      s : real;
      t : real
    );
      u : real;
    begin
    end P;
  end Sample;
end M.
```

At first rule 3 will be executed, where **ObjectsIn** will be a list containing the objects **Sample** and **r**, later denoted as `Objects(r, Sample)`. The result of this rule will be that **ObjectsOut** is assigned the list `Objects(P,r,Sample)`. The next rule will be number 1, where **Formals:ObjectsIn** will be assigned the list `Objects(P)`. After this the rules for **Formals** can be evaluated, and they will set **Formals:ObjectsOut** to `Objects(t,s,P)`, this result is then propagated to the procedure body by rule 2.

After the object lists have been constructed by rules 1-3, the environment for the formal parameters and the procedure body are constructed by rules 4 and 5, yielding:

```
Formals:Env      Env(Objects(t,s,P))
```

```
                    Env(Objects(P,r,Sample))
                    Env(Objects(Sample,M))
Body:Env            LocalEnv(Objects(Objects(u,t,s,P))
                    Env(Objects(Objects(t,s,P))
                    Env(Objects(P,r,Sample))
                    Env(Objects(Sample,M))
```

It may seem strange that some objects exists in more than one environment, but that is done to simplify the detection of illegal redeclarations of variables.

When all the rules in `pal.cg` have been evaluated, the resulting syntax tree is traversed by rules given in `analysis.puma` to ensure that variables are declared, expressions are of the proper type, actual arguments matches formal arguments, etcetera. As an example, **if** statements are checked by these rules:

```
PROCEDURE Check(Tree)
...
If(cond, then, elsifs, else) :-
  Check(cond); Check(then); Check(elsifs); Check(else);
  CheckCondition(cond);
.
...
PREDICATE CheckCondition(Tree)
  condition :- (EqualType(GetType(condition), gBooleanType)); .
  condition :- Report(Error, eIncompatibleTypes, condition); .
```

C.4 Code generation

The last phase in the compiler is to generate code for some target system. The compiler also has some unusual options to generate pretty-printed versions of the code that is easier to read than a plain text version. Each code generator is defined in a `.puma` files that starts with `code` followed by some suitable name indicating the target. As an example we can take the rules for **if** statements from `codePalsjo.puma`.

```
If(condition, then, elsifs, else) :-
  ? @if (@ Expr(condition); @) {$
    ?i Proc(then); ?d
  Proc(elsifs);
  ? @} else {$
    ?i Proc(else); ?d
  ? @}$
.
```

For the following example, it will generate C++ code that fits nicely in the **pålsjö** runtime, but would be very error prone if coded by hand.

```
ramp : input boolean;
out : output real;
...
if not ramp then
  out := 0.0;
else
  out := out + 1.0;
end if;
```

```
void Ramp::CalculateOutput() {
  if (!(*(ramp_->value))) {
    (*(out_->value)) = 0.0E+0;
    out_->Mark();
  } else {
    (*(out_->value)) = (*(out_->value)) + 1.0E+0;
    out_->Mark();
  }
}
```

Index